Response to reviewers' comments for Paper # 278

We are deeply grateful to all the reviewers for the generous and insightful suggestions which help us to improve this work to a better level. In the following, we provide explicit response to each raised concern and address our corresponding revisions in the new submission.

1. RESPONSE TO REVIEWER 1

The main con that I've spotted is the fact that the algorithm might have been more nicely framed also within the Spark environment by taking advantage of the various possibilities offered by it (e.g. caching of RDDs)

Response 1: We agree with the reviewer that utilizing advanced Spark features would benefit our solutions. Indeed, our SPARE algorithm has already taken advantages of the Spark-only features (i.e., caching of RDDs and DAG execution engine). As described in Section 5.3, our SPARE algorithm uses a run-time best-fit strategy to achieve load balance. Using the DAG execution engine, we are able to augment traditional Map-Reduce workflow to the Map-Planning-Reduce workflow. With the support of in-memory caching of RDDs, the Map results need not to be recomputed after the planning phase. As the result shown in Figure 8, SPARE with best-fit strategy saves 15% total time as compared to the randomly partition strategy (i.e., SPARE-RD). If without Spark's features, such an optimization would be hardly possible, but SPARE-RD is still more efficient than the baseline TRPM.

We do not consider utilizing other Spark extensions such as Spark-GraphX, Spark-Streaming and Spark-MLlib because these extensions, as their name suggest, are for different application scenarios.

2. RESPONSE TO REVIEW 2

The GCMP generalization is not particularly novel. Putting a maximum gap size on consecutive segments is well-known in sequence mining published more than 10 years ago.

Response 2: We admit that the gap parameter is also used in mining Gap-restricted Sequential Patterns (GSP). However, the novelty of this paper is not merely on introducing the gap-constraint to the generalized co-movement pattern. Instead, we provide a one-stop solution (both in modeling and in technical) for mining various co-movement patterns with flexible constraints. With these constraints, users are able to define a more accurate pattern which are free from the anomalies (e.g., loss-connection anomaly in Figure 2).

Besides, the essence of GSP and GCMP are different. The goal of GSP is to find the sequences which conform to the gap constraint and occur frequently. When counting the occurrence of a sequence in GSP, the object which a sequence belongs to does not matter. Therefore, in a GSP result, the involved objects may be neither distinct nor close. In contrast, GCMP imposes requirements on both the spatial closeness of objects and the number of distinct objects. Due to the unawareness of objects, techniques that are used in GSP cannot be directly applied on GCMP. We summarize this difference in Section 2.5.

In fact, I have doubts about formulating the GCMP patterns as proposed. Are we really interested in all sets of movements beyond a cardinality of size M? Take the Taxi dataset as an example. Let say that there are lots of taxis going from the airport to downtown. Let say that there are 1000 such taxis. For a given M, are we interested in $\binom{1000}{M}$ answers? So this speaks to the problem of picking M. If M is 500, what is $\binom{1000}{M}$? In fact, even if the system gives the single answer of $\binom{1000}{1000}$, I am not sure I am interested in this pattern as I already know that there are many taxis going from the airport to downtown. What I think I am really interested in are GCMP that are "anomalous", which is much harder to define.

Response 3: We appreciate the reviewer's concerns on the size of the output. In fact, both TRPM and SPARE have considered compressing the output by only discovering the patterns with larger object set. Particularly, in the Line Sweeping Mining (Algorithm 1) of TRPM, whenever a pattern c becomes valid, it is directly outputted (Lines 14-16). This prohibits outputting c's subset. Similarly, in the Apriori Enumerator (Algorithm 3) of SPARE, a pattern c is outputted if none of its supersets could become a valid pattern (Line 12-14). This ensures the output c is not a subset of some pattern. These mechanisms effectively condense the output by subsuming smaller patterns using their supersets. Linking to the Taxi example, if the ground truth contains 1000 taxis travel together, then both TRPM and SPARE tend to output the single pattern. In addition, real detection of "anomalous" co-moment pattern needs to impose stringent constraints via M, K, L, G. This would further reduces the size of the output.

Regarding the second weak point, one line of related work is the superimposition of constraints on spatiotemporal mining. An example is a road network. In other words, given a road network, the network imposes constraints on GCMP.

Response 4: We understand that spatiotemporal mining with domain-constraints looks similar to GCMP. However, those techniques often heavily utilize the domain knowledge (such as directions in road networks) which are nontrivial to be generalized. Differently, since our GCMP model and technical solutions do not leverage the domain knowledge, we can support pattern discovery in broader scenarios such as user check-in histories in social networks, visitor movements in a building and passenger flows in a city. We also add the discussion in Section 2.3.

3. RESPONSES TO REVIEWER 3

Though the implementation references Spark as the implementation platform (as it also reflects in github repo provided), the algorithm design is mostly limited to MapReduce, aka only Hadoop, which is a very small subset of Spark. This may have a negative impact on the baseline implementation. Particularly, recent releases of Spark have introduced window functions that can be applied directly in the sliding window scenario here. Certainly, the algorithm has to be redesigned to use DataFrame (and/or Spark SQL) interface, it has been noted that this is a very efficient way to execute window functions in Spark

Response 5: We thank the reviewer for suggesting a better way of implementing the baseline algorithm (i.e., TRPM). However, this could hardly be done after our investigations. The major reason is that the current version of Spark-SQL (i.e., 1.6.2 just released in June) does not support the User Defined Aggregate Function (UDAF) on window function [1]. Without UDAF, implementing our Line Sweep algorithm (Algorithm 1) using Spark-SQL primitive aggregates (e.g., sum, avg, rank and nth-tile) is beyond our abilities.

Further, we believe the performance boost on TRPM from Spark-SQL is very limited. The reason is that our Line Sweep Mining algorithm (Algorithm 1) is not a properly reducible function. That is, the result of a smaller window could not be used to compute the result of a larger window. In such a case, Spark-SQL has to process each window independently in parallel, which is equivalent to our TRPM implementation.

In particular, it would be great to provide the difference in the number of partitions/splits, the amount of processing and memory usage (i.e., vcore and memory seconds) between TRPM and SPARE

Response 6: In our implementation, we take a fixed 486 partitions (Section 6) for each RDD. The partition number is determined based on the parallel tuning guide by Cloudera [2]. To study the load balance, we revise Section 6.2.2 by comparing TRPM, SPARE with random task allocation and SPARE with best-fit task allocation (Section 5.3). We present the summary of the execution time for each method on longest job, shortest job and standard deviation of all jobs in Table 9. As the table show, SPARE has a better load balance (low deviation) than SPARE-RD which indicates the effectiveness of the best-fit strategy. Further, TRPM generally keeps a low deviation. This is because that TRPM always adopts the equal-size (i.e. η) partition. Nevertheless, both the longest and shortest jobs in TRPM are higher than SPARE based solutions.

Next, we add Table 7 (Section 6.1) to compare the resource usage (e.g., Vcore-seconds, Memory and Running Time) between SPARE and TRPM. The table tells that both SPARE and TRPM are resource efficient. In particular, the RDDs for both SPARE and TRPM only take less than 20% of the available memory. This infers the potential of SPARE and TRPM in handling even larger trajectories with current system resources.

A plot that breaks down the performance gain by each method would be greatly appreciated by the readers.

Response 7: We complement Figure 8 (Section 6.1) by including the breakdown cost of TRPM. In SPARE, star partition takes place in the Map-Shuffle phase while Apriori enumeration takes place in the Reduce phase. As shown in the new Figure 8, both star partition and apriori enumeration contribute to the final performance gain of SPARE. In particular, in the Map-Shuffle phase, SPARE saves 56%-73% in time as compared to TRPM. This indicates that less data are transformed and shuffled in the star partition. In the Reduce phase, SPARE saves 46%-81% time as compared to TRPM. This confirms the efficiency of our Apriori enumerator with various optimization (e.g., sequence simplification, anti-monotonicity and forward closure checking).

Some choices of words may need to be reconsidered: for example, "a bunch of" might not be appropriate in a technical paper.

Response 8: We thank the reviewer for the correction. We have changed these unprofessional terms.

References to star partitioning and apriori pruning are missing. Though these are well known, they need to clearly cited. At least the following reference is missing:

Response 9: We add the corresponding references in Section 5.1 and Section 5.2.

In "In contrast, when utilizing the multicore environment, SPAREP achieves 7 times speedup and SPARES achieves 10 times speedup.", was "multicore" referring to the use of all 16 cores in one of your node? The specification of the machine was not clear.

Response 10: We use all cores in the single node to conduct the experiment. We revise the description in Section 6.2.3.

[1]JIRA Spakr-8641: Native Spark Window Functions `https://issues.apache.org/jira/browse/SPARK-8641`

[2]Spark Performance Tuning `http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/`

The computation of "eta" was slightly different than that in the paper

Response 11: We have updated the GitHub repository to rectify the typo in the equation.

4. RESPONSE TO META REVIEWER

Novelty: The GCMP generalization is not particularly novel. Please elaborate on the novelty of the work.

Response 12: In addition to the points in Response 2, another novelty of our work lies in the techniques. Although the idea of star partition and apriori enumeration are well-known, linking them together to solve the problem in trajectory domain has not been attempted before. Besides, we input heavy details on these algorithms (e.g., theoretical bounds of partition and anti-monotonicity) which have not been previously proposed.

Parameter setting: Are we really interested in all sets of movements beyond a cardinality of size M? Please elaborate on the motivation.

Response 13: Similar responses with Response 3.

Spark implementation: the paper references Spark as the implementation platform but the algorithm is limited to MapReduce. Spark has capabilities beyond MapReduce such as window functions. Please provide an implementation that uses the relevant features of Spark, or a convincing discussion of how these features can be useful and why they were not used.

Response 14: Similar responses with Response 1 and 5.

More details in the performance evaluation: Please provide more details about data partitioning and the effect of skew. Also provide details about how star partitioning and a priori pruning contribute to performance. Please provide references to these two methods.

Response 15: Similar responses with Response 6, 7 and 9.